Lecture 2 (part2)

Topics covered:
Instruction Set Architecture

Instruction execution and sequencing

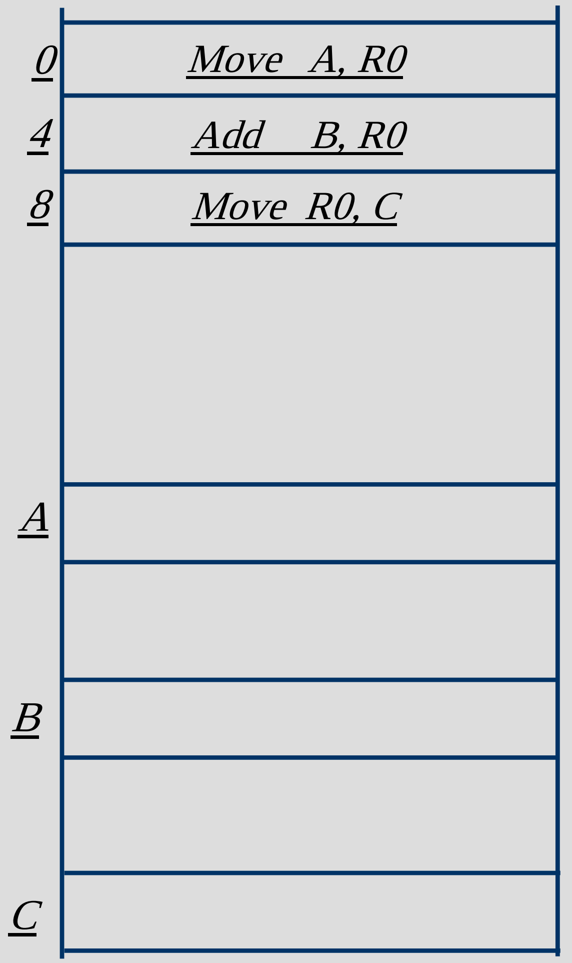
- Recall the fetch/execute cycle of instruction execution.
- In order to complete a meaningful task, a number of instructions need to be executed.
- During the fetch/execution cycle of one instruction, the Program Counter (PC) is updated with the address of the next instruction:
  - PC contains the address of the memory location from which the next instruction is to be fetched.
- When the instruction completes its fetch/execution cycle, the contents of the PC point to the next instruction.
- Thus, a sequence of instructions can be executed to complete a task.

Instruction execution and sequencing (contd..)

- Simple processor model
- Processor has a number of general purpose registers.
- Word length is 32 bits (4 bytes).
- Memory is byte addressable.
- Each instruction is one word long.
- Instructions allow one memory operand per instruction.
  - One register operand is allowed in addition to one memory operand.
- Simple task:
  - Add two numbers stored in memory locations A and B.
  - Store the result of the addition in memory location C.

```
Move A, R0   (Move the contents of location A to register R0)
Add  B, R0   (Add the contents of location B to register R0)
Move R0, C   (Move the contents of register R0 to location C)
```

Instruction execution and sequencing (contd..)



Execution steps:

Step I:

- *-PC holds address 0.*
- *-Fetches instruction at address 0.*
- *-Fetches operand A.*
- *-Executes the instruction.*
- *-Increments PC to 4.*

Step II:

- *-PC holds address 4.*
- *-Fetches instruction at address 4.*
- *-Fetches operand B.*
- *-Executes the instruction.*
- *-Increments PC to 8.*

Step III:

- *-PC holds address 8.*
- *-Fetches instruction at address 8.*
- *-Executes the instruction.*
- *-Stores the result in location C.*

Instructions are executed one at a time in order of increasing addresses.
"Straight line sequencing"

Instruction execution and sequencing (contd..)

- Consider the following task:
  - Add 10 numbers.
  - Number of numbers to be added (in this case 10) is stored in location N.
  - Numbers are located in the memory at NUM1, …. NUM10
  - Store the result in SUM.

```
Move NUM1,  R0    (Move the contents of location NUM1  to register R0)
Add   NUM2, R0    (Add the contents of location NUM2 to register R0)
Add   NUM3, R0    (Add the contents of location NUM3 to register R0)
Add   NUM4, R0    (Add the contents of location NUM4 to register R0)
Add   NUM5, R0    (Add the contents of location NUM5 to register R0)
Add   NUM6, R0    (Add the contents of location NUM6 to register R0)
Add   NUM7, R0
Add   NUM8, R0
Add   NUM9, R0
Add   NUM10, R0
Move  R0, SUM     (Move the contents of register R0 to location SUM)
```

Instruction sequencing and execution (contd..)

- Separate Add instruction to add each number in a list, leading to a long list of Add instructions.
- Task can be accomplished in a compact way, by using the Branch instruction.

```
        Move N, R1   (Move the contents of location N, which is the number
                     of numbers to be added to register R1)
        Clear R0     (This register holds the sum as the numbers are added)
LOOP    Determine the address of the next number.
        Add the next number to R0.
        Decrement R1 (Counter which indicates how many numbers have been
                     added so far).
        Branch>0  LOOP  (If all the numbers haven't been added, go to LOOP)
        Move R0, SUM
```

Instruction execution and sequencing (contd..)

- *Decrement R1*:
  - Initially holds the number of numbers that is to be added (*Move N, R1*).
  - Decrements the count each time a new number is added (*Decrement R1*).
  - Keeps a count of the number of the numbers added so far.
- *Branch>0 LOOP*:
  - Checks if the count in register R1 is 0 (*Branch > 0*)
  - If it is 0, then store the sum in register R0 at memory location SUM (*Move R0, SUM*).
  - If not, then get the next number, and repeat (*go to LOOP*). Go to is specified implicitly.
- Note that the instruction (*Branch > 0 LOOP*) has no explicit reference to register R1.

Instructions execution and sequencing (contd..)

- Processor keeps track of the information about the results of previous operation.
- Information is recorded in individual bits called "condition code flags". Common flags are:
  - N (negative, set to 1 if result is negative, else cleared to 0)
  - Z (zero, set to 1 if result is zero, else cleared to 0)
  - V (overflow, set to 1 if arithmetic overflow occurs, else cleared)
  - C (carry, set to 1 if a carry-out results, else cleared)
- Flags are grouped together in a special purpose register called "condition code register" or "status register".

If the result of Decrement R1 is 0, then flag Z is set.
Branch> 0, tests the Z flag.
If Z is 1, then the sum is stored.
Else the next number is added.

Instruction execution and sequencing (contd..)

- Branch instructions alter the sequence of program execution
  - Recall that the PC holds the address of the next instruction to be executed.
  - Do so, by loading a new value into the PC.
  - Processor fetches and executes instruction at this new address, instead of the instruction located at the location that follows the branch.
  - New address is called a "branch target".
- Conditional branch instructions cause a branch only if a specified condition is satisfied
  - Otherwise the PC is incremented in a normal way, and the next sequential instruction is fetched and executed.
- Conditional branch instructions use condition code flags to check if the various conditions are satisfied.

Instruction sequencing and execution (contd..)

- How to determine the address of the next number?
- Recall the addressing modes:
  - Initialize register R2 with the address of the first number using Immediate addressing.
  - Use Indirect addressing mode to add the first number.
  - Increment register R2 by 4, so that it points to the next number(word-length=4 bytes).

```
        Move N, R1
        Move #NUM1, R2 (Initialize R2 with address of NUM1)
        Clear R0
LOOP    Add (R2), R0        (Indirect addressing)
        Add #4, R2          (Increment R2 to point to the next number)
        Decrement R1
        Branch>0 LOOP
        Move R0, SUM
```

Instruction execution and sequencing (contd..)

- Note that the same can be accomplished using "autoincrement mode":

```
            Move N, R1
            Move #NUM1, R2 (Initialize R2 with address of NUM1)
            Clear R0
LOOP        Add (R2)+, R0        (Autoincrement)
            Decrement R1
            Branch>0 LOOP
            Move R0, SUM
```

Numbers Notation

Decimal number	ADD #93,R1
Binary number	ADD #%01011101,R1
Hexadecimal number	ADD #$5D,R1

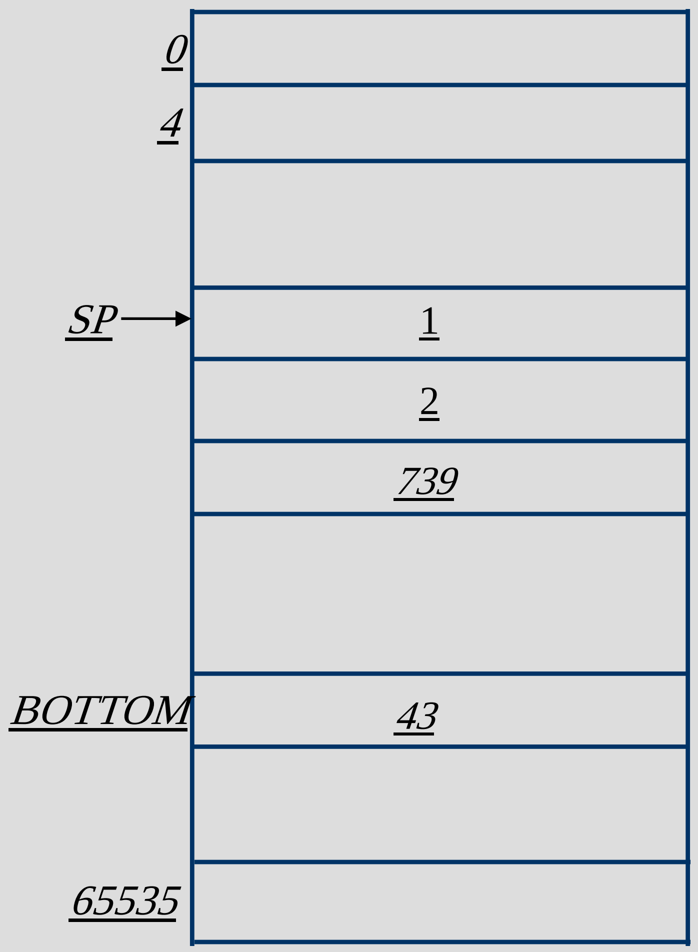
Stacks

- A stack is a list of data elements, usually words or bytes with the accessing restriction that elements can be added or removed at one end of the stack.
  - End from which elements are added and removed is called the "top" of the stack.
  - Other end is called the "bottom" of the stack.
- Also known as:
  - Pushdown stack.
  - Last in first out (LIFO) stack.
- *Push* - placing a new item onto the stack.
- *Pop* - Removing the top item from the stack.

Stacks (contd..)

- Data stored in the memory of a computer can be organized as a stack.
  - Successive elements occupy successive memory locations.
- When new elements are pushed on to the stack they are placed in successively lower address locations.
  - Stack grows in direction of decreasing memory addresses.
- A processor register called as "Stack Pointer (SP)" is used to keep track of the address of the element that is at the top at any given time.
  - A general purpose register could serve as a stack pointer.

Stacks (contd..)



- Processor with 65536 bytes of memory.
- Byte addressable memory.
- Word length is 4 bytes.
- First element of the stack is at BOTTOM.
- SP points to the element at the top.

- Push operation can be implemented as:

```
    Subtract #4, SP
    Move A, (SP)
```

- Pop operation can be implemented as:

```
    Move (SP), B
    Add #4, SP
```

- Push with autodecrement:

```
    Move A, -(SP)
```

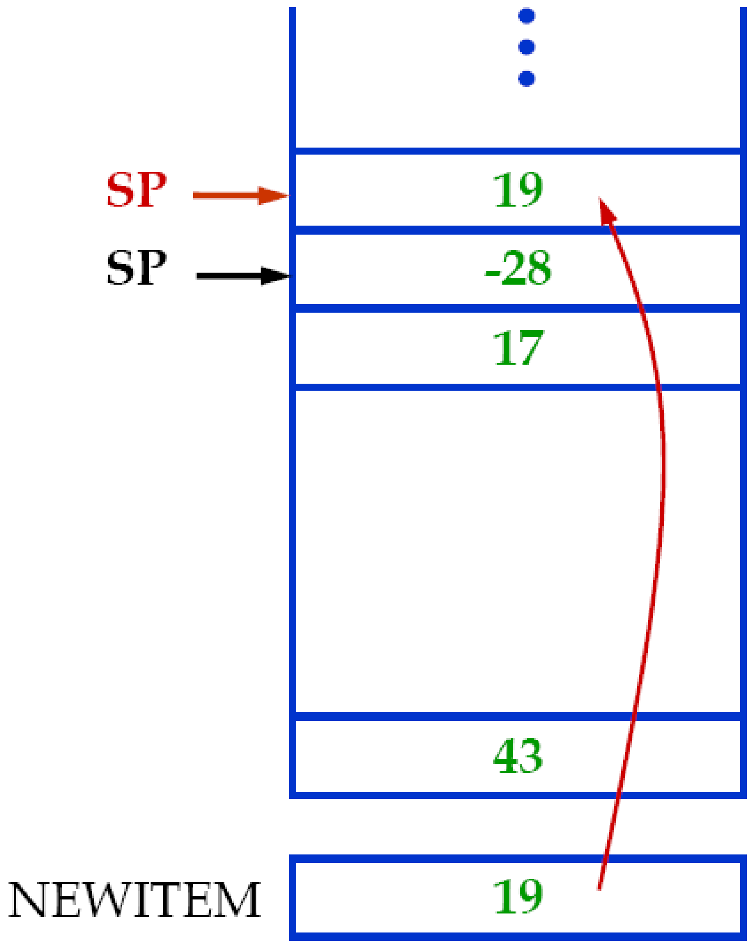
- Pop with autoincrement:

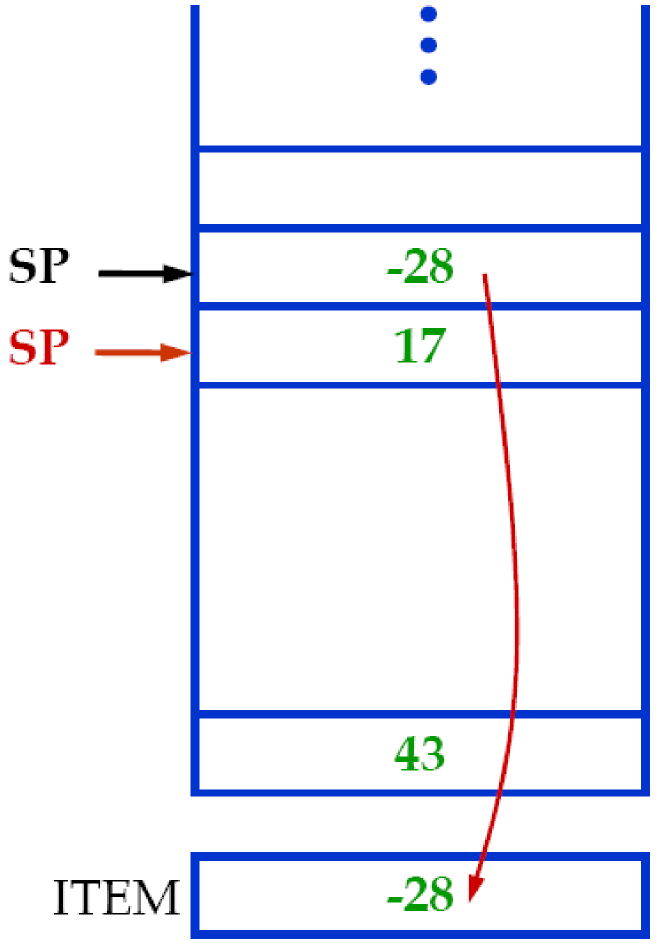
```
    Move (SP)+, A
```

Push and Pop Operations

Move NEWITEM, -(SP)

Move (SP)+, ITEM





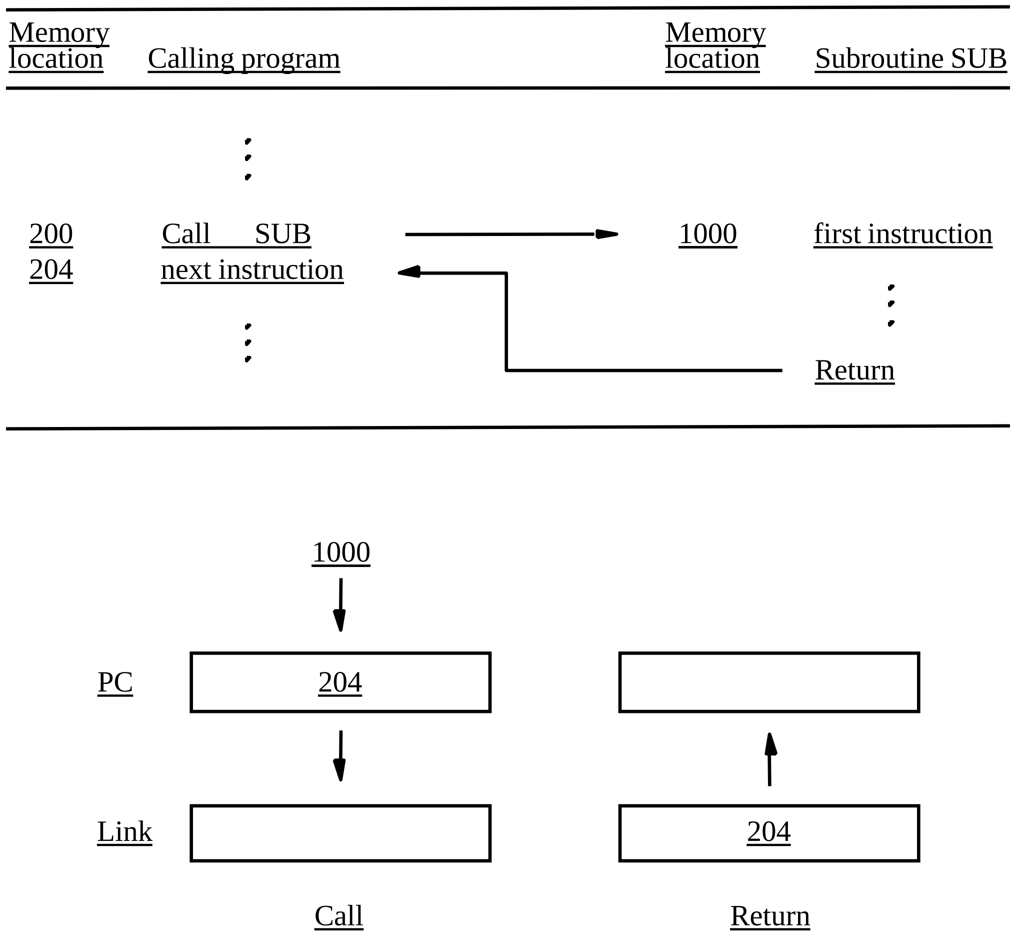
Subroutines

- In a program subtasks that are repeated on different data values are usually implemented as subroutines.
- When a program requires the use of a subroutine, it branches to the subroutine.
  - Branching to the subroutine is called as "calling" the subroutine.
  - Instruction that performs this branch operation is ***Call***.
- After a subroutine completes execution, the calling program continues with executing the instruction immediately after the instruction that called the subroutine.
  - Subroutine is said to "return" to the program.
  - Instruction that performs this is called ***Return***.
- Subroutine may be called from many places in the program.
  - How does the subroutine know which address to return to?

Subroutines (contd..)

- Recall that when an instruction is being executed, the PC holds the address of the next instruction to be executed.
  - This is the address to which the subroutine must return.
  - This address must be saved by the Call instruction.
- Way in which a processor makes it possible to call and return from a subroutine is called "subroutine linkage method".
- The return address could be stored in a register called as "Link register"
- Call instruction:
  - Stores the contents of the PC in a link register.
  - Branches to the subroutine.
- Return instruction:
  - Branch to the address contained in the link register.

Subroutines (contd..)

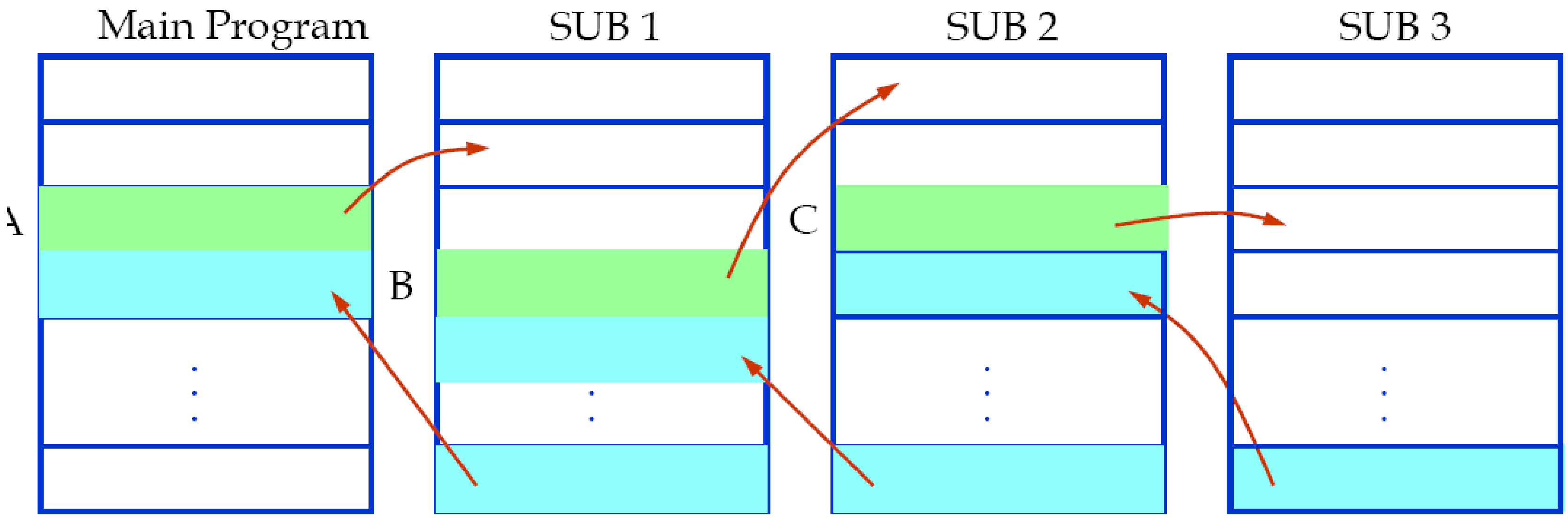


- *Calling program calls a subroutine, whose first instruction is at address 1000.*
- *The Call instruction is at address 200.*
- *While the Call instruction is being executed, the PC points to the next instruction at address 204.*
- *Call instructions stores address 204 in the Link register, and loads 1000 into the PC.*
- *Return instruction loads back the address 204 from the link register into the PC.*

Subroutines and stack

- For nested subroutines:
  - After the last subroutine in the nested list completes execution, the return address is needed to execute the return instruction.
  - This return address is the last one that was generated in the nested call sequence.
  - Return addresses are generated and used in a "Last-In-First-Out" order.
- Push the return addresses onto a stack as they are generated by subroutine calls.
- Pop the return addresses from the stack as they are needed to execute return instructions.

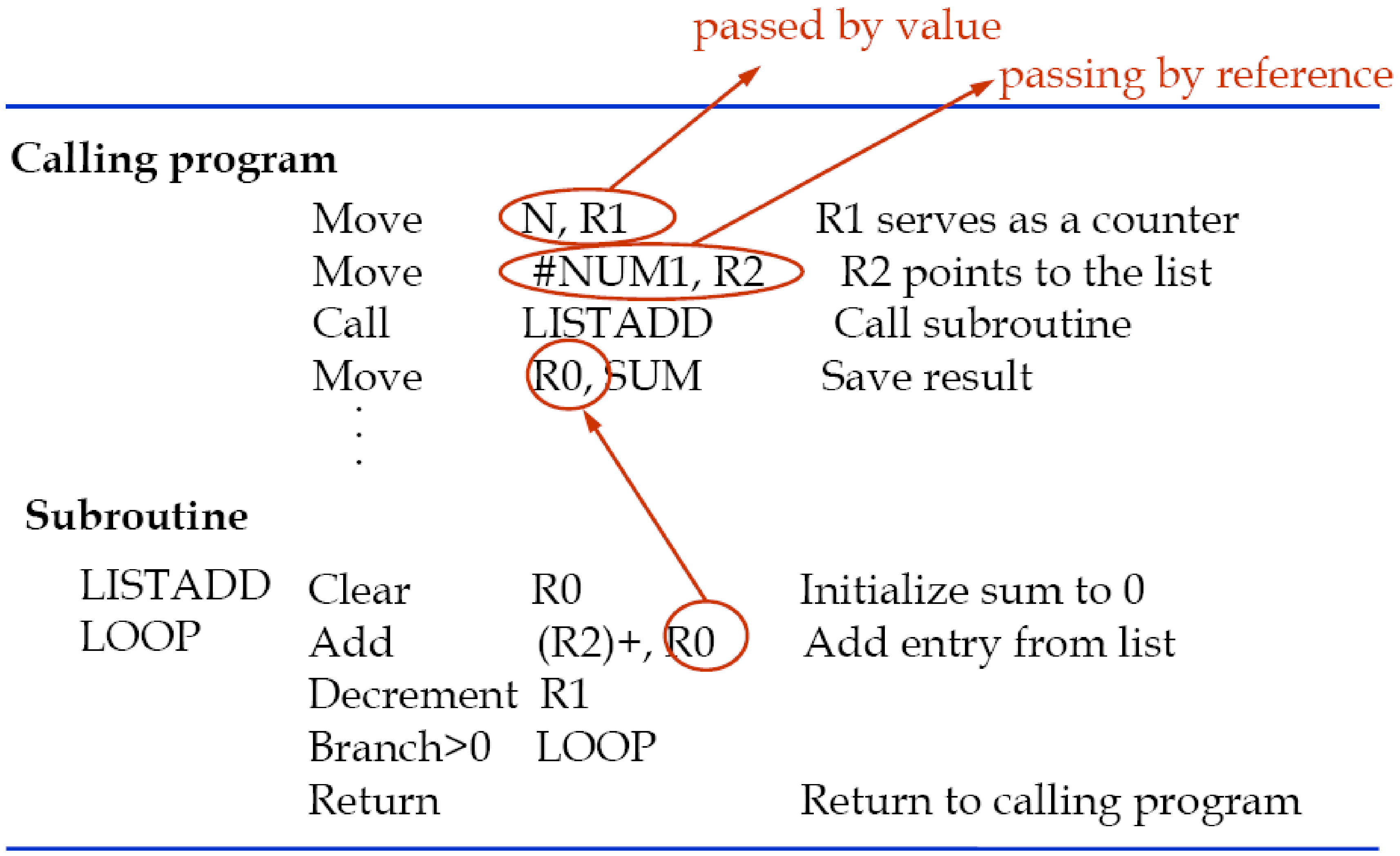
Example of Subroutine Nesting



Parameter Passing

- When calling a subroutine, a program must provide to the subroutine
  - the **parameters**, that is, the **operands** or their addresses, to be used in the computation.
  - Later, the subroutine **returns** other parameters, in this case, the **result of computation**
- The exchange of information between a calling program and a subroutine is referred to as parameter passing
- Parameter passing approaches
  - The parameters may be placed in registers or in memory locations, where they can be accessed by the subroutine
  - The parameters may be placed on the processor stack used for saving the return address

Passing Parameters with Registers



Calling program

```
                Move        N, R1          R1 serves as a counter
                Move        #NUM1, R2      R2 points to the list
                Call        LISTADD        Call subroutine
                Move        R0, SUM        Save result
                  .
                  .
                  .
```

Subroutine

```
    LISTADD     Clear       R0             Initialize sum to 0
    LOOP        Add         (R2)+, R0      Add entry from list
                Decrement   R1
                Branch>0    LOOP
                Return                     Return to calling program
```

Shift Instructions

- Many applications (multiplication, division, substring, …) require the bits of an operand to be shifted right or left some positions.
- Logical Shift
  - Shifts an operand to the left (or right) over a number of bit positions specified in a count operand of the instruction
  - Zero bits are brought to the vacated positions at the right (or left) end of the destination operand
  - Bits shifted out are passed to the carry flag C, and then dropped out.

```
LshiftL    count, dest
LshiftR    count, dest
```

Shift Instructions

- Arithmetic Shift(preserves the sign of the number)
  - Shifting a number one bit position to the left is equivalent to multiplying it by 2.
  - Shifting it to the right is equivalent to dividing it by 2.
  - Overflow might occur during shift left
  - The remainder is lost in shifting right
  - During shifting right, the sign bit must be repeated as the fill in bit for the vacated position

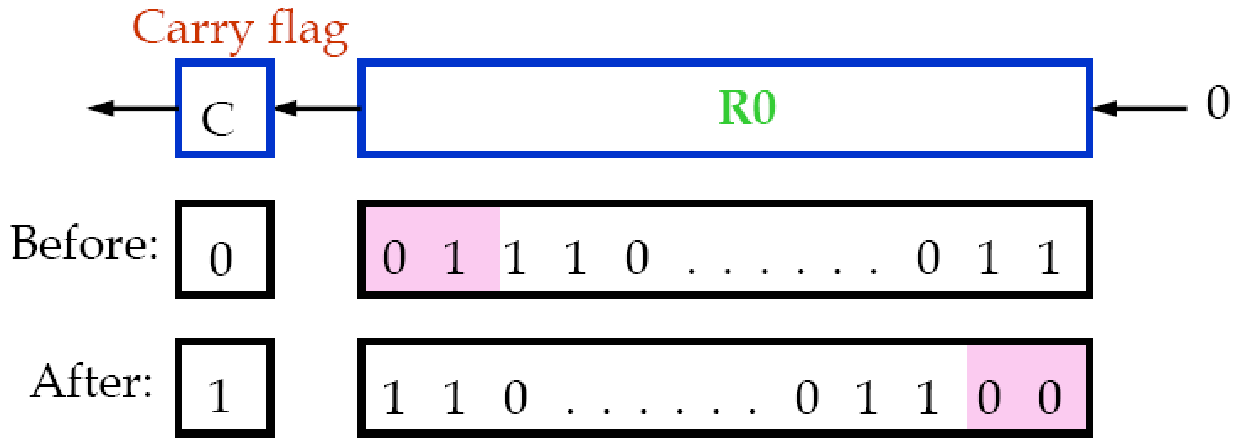
```
AShiftL    count, dest
ASiftR     count, dest
```

Shift Instructions

- Logical shifts

Logic shift left

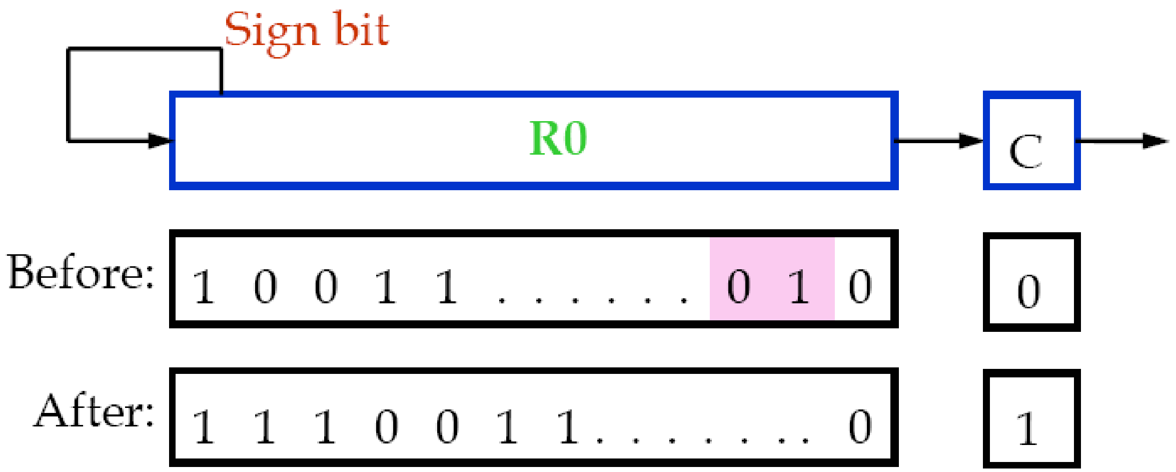
LShiftL #2, R0



- Arithmetic shifts

shift right

AShiftR #2, R0



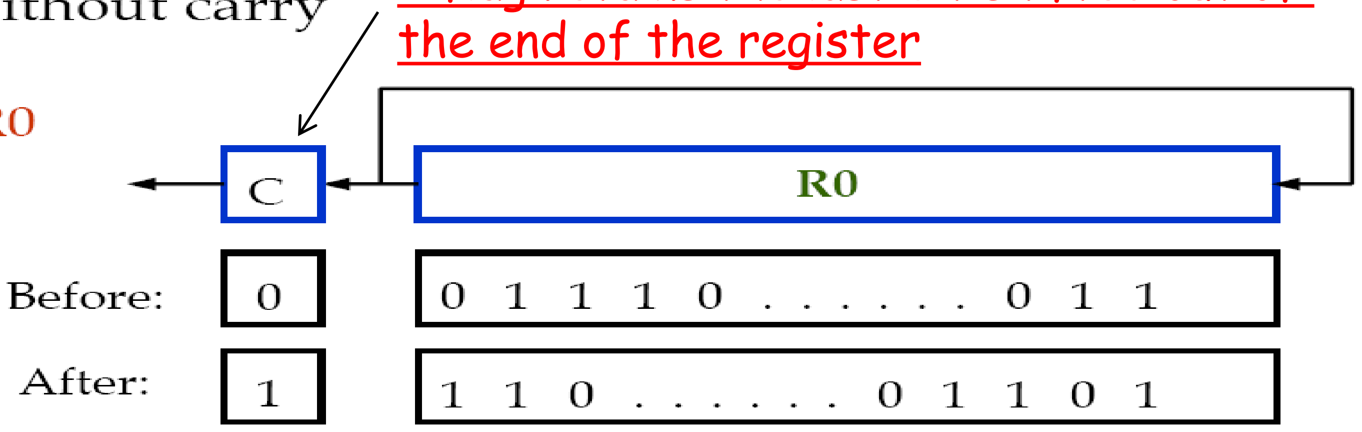
Rotate Instructions

To preserve all bits during the shift operation, we use the Rotate Instructions.

- Rotate left without carry

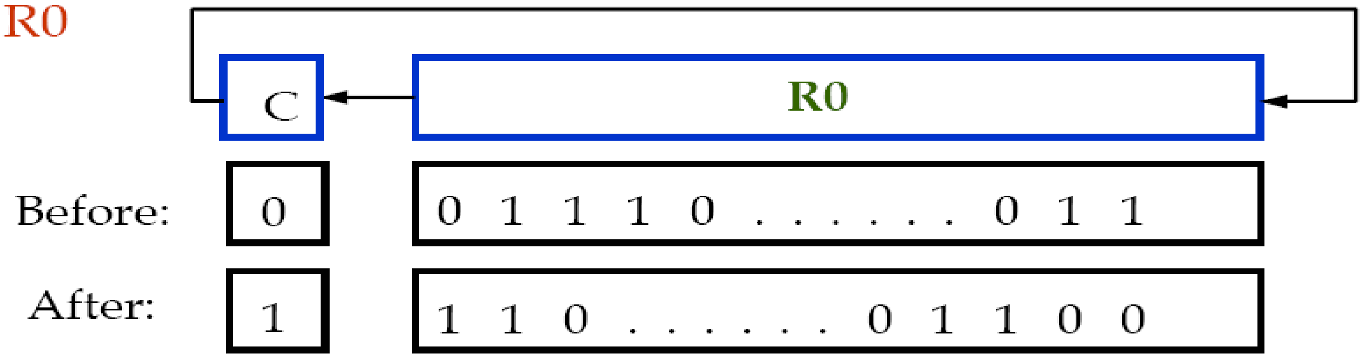
RotateL #2, R0

C flag retains the last bit shifted out of the end of the register



- Rotate left with carry

RotateLC #2, R0



Logical Operations

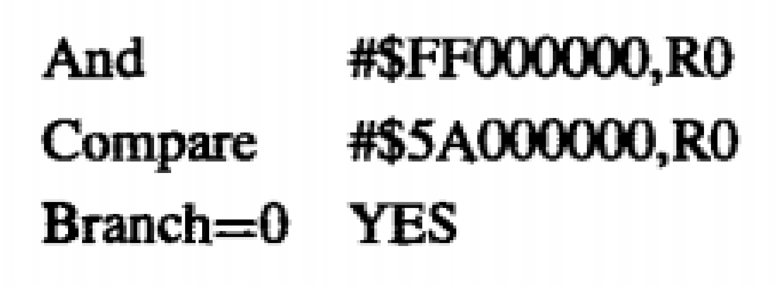
- Not dst

To obtain two's complement of a number contained in R0

```
Not R0
Add #1,R0
```

- AND
- OR

R0 contains four ASCII characters, we wish to determine if the left most character is Z (=5A in hex)

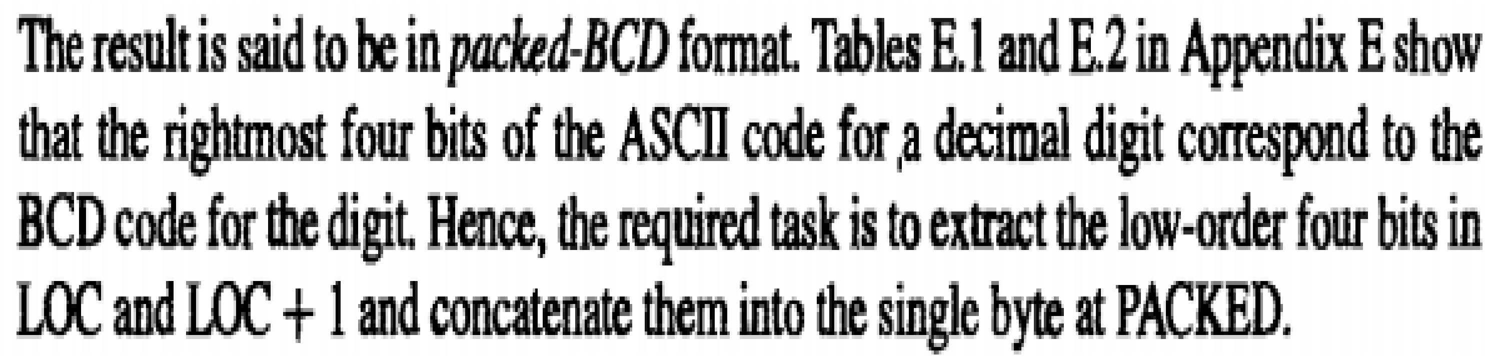


```
And          #$FF000000,R0
Compare      #$5A000000,R0
Branch=0     YES
```

Digit Packing Example

- Digit-Packing

2 decimal digits (two bytes) represented in ASCII code located at memory locations (Loc, Loc+1), we wish to represent them in BCD code and store them in a single byte location (Packed).


The result is said to be in *packed-BCD* format. Tables E.1 and E.2 in Appendix E show that the rightmost four bits of the ASCII code for a decimal digit correspond to the BCD code for the digit. Hence, the required task is to extract the low-order four bits in LOC and LOC + 1 and concatenate them into the single byte at PACKED.

❑Program:

```
Move      #Loc, R0          Ro points to first digit
MoveByte    (R0)+, R1       Load First Byte into R1
LShiftL #4,R1               Shift Left by 4 bit positions
MoveByte    (R0), R2        Load Second Byte into R2
And       #$F, R2           Eliminate high-order bits of R2
Or        R1, R2            Concatenate both BCD digits
MoveByte    R2, Packed      Store the result into Packed
```

We assume in this program that MoveByte instruction pushes the moved byte into the right most byte of the registers R1, R2.

Multiplication and Division

- Two signed integers can be multiplied or divided as:

  Multiply Ri, Rj

  Which performs the operation

  Rj ← [Ri] x [Rj]

  The product of two n-bits numbers can be large as 2n bits.

  Some processors produce the product in 2 adjacent registers Rj, Rj+1.

- **Divide Ri, Rj**

  which performs the operation:

  Rj ← [Rj] / [Ri]

  Placing the quotient in Rj, the remainder in Rj+1

- Computers that do not have Multiply and Divide instructions can use Add, Subtract, Shift, and Rotate instructions.

Example Program

- Compute the dot product of two vectors A, B of n-bits using:

  Dot Product = ∑ A(i) x B(i)

- The Program:

```
        Move      #Avec, R1        R1 points to vector A
        Move      #Bvec, R2        R2 points to vector B
        Move      N, R3            R3 serves as the vector size
        Clear     R0               R0 accumulates the dot product
   Loop Move      (R1)+, R4        load the first number into R4
        Multiply  (R2)+, R4        Computes the product
        Add       R4, R0           Add to previous Sum
        Decrement R3               Decrement the vector Size
        Branch>0  Loop             Loop again if not done
        Move      R0, DotProduct   Store the result into Memory
```

Assembly language

- Recall that information is stored in a computer in a binary form, in a patterns of 0s and 1s.
- Symbolic names are used to represent patterns.
  - So far we have used normal words such as *Move, Add, Branch,* to represent corresponding binary patterns.
- When we write programs for a specific computer, the normal words need to be replaced by acronyms called mnemonics.
  - E.g., *MOV, ADD, INC*
- A complete set of symbolic names and rules for their use constitute a programming language, referred to as the assembly language.

Assembly language (contd..)

- Programs written in assembly language need to be translated into a form understandable by the computer, namely, binary, or machine language form.
- Translation from assembly language to machine language is performed by an assembler.
  - Original program in assembly language is called source program.
  - Assembled machine language program is called object program.
- Each mnemonic represents the binary pattern, or *OP code* for the operation performed by the instruction.
- Assembly language must also have a way to indicate the addressing mode being used for operand addresses.
- Sometimes the addressing mode is indicated in the OP code mnemonic.
  - E.g., ***ADDI*** may be a mnemonic to indicate an addition operation with an immediate operand.

Assembly language (contd..)

- Assembly language allows programmer to specify other information necessary to translate the source program into an object program.
  - How to assign numerical values to the names.
  - Where to place instructions in the memory.
  - Where to place data operands in the memory.
- The statements which provide additional information to the assembler to translate source program into an object program are called assembler directives or commands.

Assembly language (contd..)

	100	Move	N,R1
	104	Move	#NUM1,R2
	108	Clear	R0
LOOP	112	Add	(R2),R0
	116	Add	#4,R2
	120	Decrement	R1
	124	Branch>0	LOOP
	128	Move	R0,SUM
	132		
		⋮	
SUM	200		
N	204	100	
NUM1	208		
NUM2	212		
		⋮	
NUM*100*	604		

- *What is the numeric value assigned to SUM?*
- *What is the address of the data NUM1 through NUM100?*
- *What is the address of the memory location represented by the label LOOP?*
- *How to place a data value into a memory location?*

Assembly language (contd..)

	Memory address label	Operation	Addressing or data information
Assembler directives	SUM	EQU	200
		ORIGIN	204
	N	DATAWORD	100
	NUM1	RESERVE	400
		ORIGIN	100
Statements that generate machine instructions	START	MOVE	N,R1
		MOVE	#NUM1,R2
		CLR	R0
	LOOP	ADD	(R2),R0
		ADD	#4,R2
		DEC	R1
		BGTZ	LOOP
		MOVE	R0,SUM
Assembler directives		RETURN	
		END	START

EQU:
- *Value of SUM is 200.*

ORIGIN:
- *Place the datablock at 204.*

DATAWORD:
- *Place the value 100 at 204*
- *Assign it label N.*
- *N EQU 100*

RESERVE:
- *Memory block of 400 words is to be reserved for data.*
- *Associate NUM1 with address 208*

ORIGIN:
- *Instructions of the object program to be loaded in memory starting at 100.*

RETURN:
- *Terminate program execution.*

END:
- *End of the program source text*

Assembly language (contd..)

- Assembly language instructions have a generic form:

  Label Operation Operand(s) Comment

- Four fields are separated by a delimiter, typically one or more blank characters.
- Label is optionally associated with a memory address:
  - May indicate the address of an instruction to be executed.
  - May indicate the address of a data item.
- How does the assembler determine the values that represent names?
  - Value of a name may be specified by EQU directive.
    - SUM EQU 100
  - A name may be defined in the Label field of another instruction, value represented by the name is determined by the location of that instruction in the object program.
    - E.g., BGTZ LOOP, the value of LOOP is the address of the instruction ADD (R2) R0

Assembly language (contd..)

- Assembler scans through the source program, keeps track of all the names and corresponding numerical values in a "symbol table".
  - When a name appears second time, it is replaced with its value from the table.
- What if a name appears before it is given a value, for example, branch to an address that hasn't been seen yet (forward branch)?
  - Assembler can scan through the source code twice.
  - First pass to build the symbol table.
  - Second pass to substitute names with numerical values.
  - Two pass assembler.

Encoding of machine instructions

- Instructions specify the operation to be performed and the operands to be used.
- Which operation is to be performed and the addressing mode of the operands may be specified using an encoded binary pattern referred to as the "OP code" for the instruction.
- Consider a processor with:
  - Word length 32 bits.
  - 16 general purpose registers, requiring 4 bits to specify the register.
  - 8 bits are set aside to specify the OP code.
  - 256 instructions can be specified in 8 bits.

Encoding of machine instructions (contd..)

One-word instruction format.

8	7	7	10
OP code	Source	Dest	Other info

Opcode	:	*8 bits.*
Source operand	:	*4 bits to specify a register*
		3 bits to specify the addressing mode.
Destination operand	:	*4 bits to specify a register.*
		3 bits to specify the addressing mode.
Other information	:	*10 bits to specify other information*
		such as index value.

Encoding of machine instructions (contd..)

What if the source operand is a memory location specified using the absolute addressing mode?

8	3	7	14
OP code	Source	Dest	

Opcode	:	*8 bits.*
Source operand	:	*3 bits to specify the addressing mode.*
Destination operand	:	*4 bits to specify a register.*
		3 bits to specify the addressing mode.

- *Leaves us with 14 bits to specify the address of the memory location.*
- *Insufficient to give a complete 32 bit address in the instruction.*
- *Include second word as a part of this instruction, leading to a two-word instruction.*

Encoding of machine instructions (contd..)

Two-word instruction format.

OP code	Source	Dest	Other info
Memory address/Immediate operand			

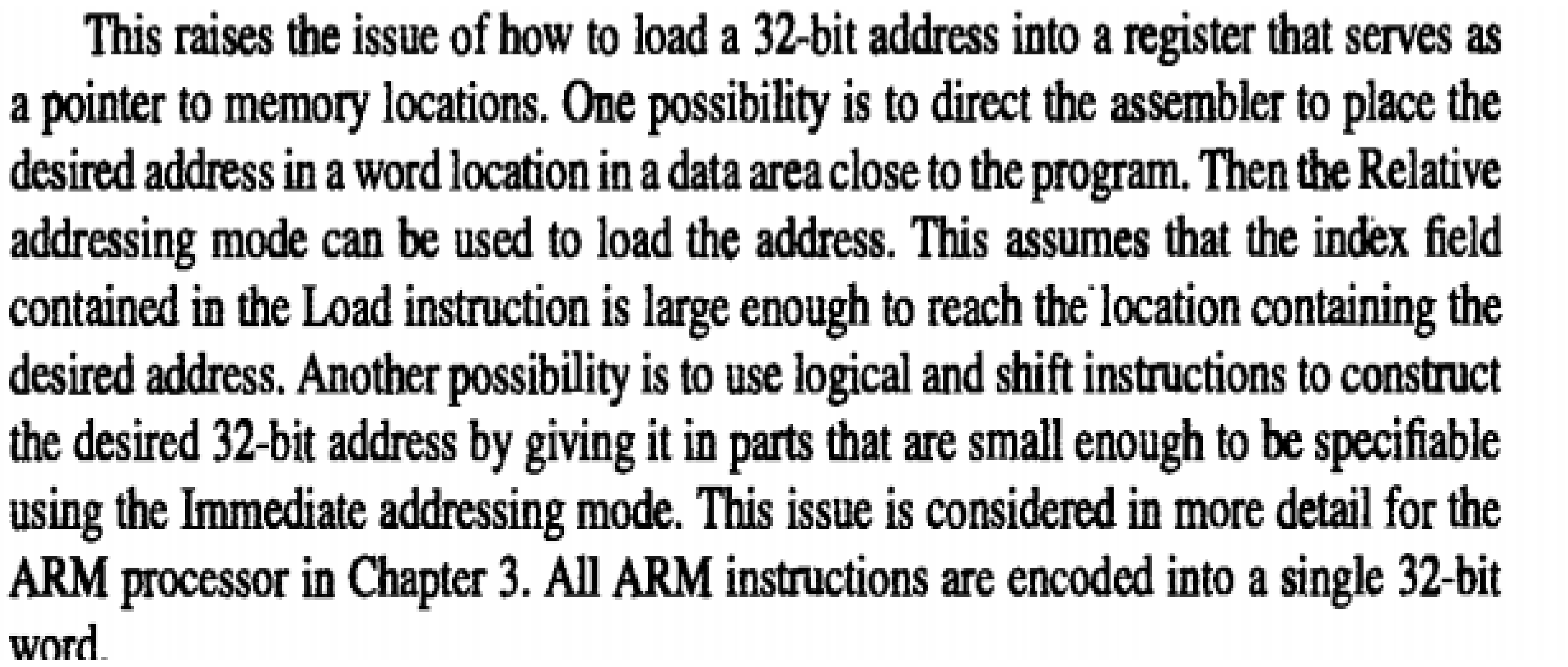
- *Second word specifies the address of a memory location.*
- *Second word may also be used to specify an immediate operand.*

- Complex instructions can be implemented using multiple words.
- *Complex Instruction Set Computer (CISC)* refers to processors using instruction sets of this type.

Encoding of machine instructions (contd..)

- Insist that all instructions must fit into a single 32 bit word:
  - Instruction cannot specify a memory location or an immediate operand.
  - *ADD R1, R2* can be specified.
  - *ADD LOC, R2* cannot be specified.
  - Use indirect addressing mode: *ADD (R3), R2*
  - R3 serves as a pointer to memory location LOC.
- How to load address of LOC into R3?
  - Relative addressing mode.

❑How to load address of LOC into R3?

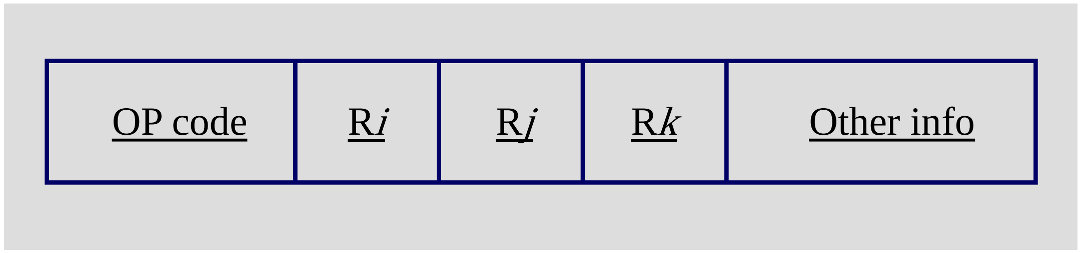


This raises the issue of how to load a 32-bit address into a register that serves as a pointer to memory locations. One possibility is to direct the assembler to place the desired address in a word location in a data area close to the program. Then the Relative addressing mode can be used to load the address. This assumes that the index field contained in the Load instruction is large enough to reach the location containing the desired address. Another possibility is to use logical and shift instructions to construct the desired 32-bit address by giving it in parts that are small enough to be specifiable using the Immediate addressing mode. This issue is considered in more detail for the ARM processor in Chapter 3. All ARM instructions are encoded into a single 32-bit word.

Encoding of machine instructions (contd..)

- Restriction that an instruction must occupy only one word has led to a style of computers that are known as *Reduced Instruction Set Computers (RISC).*
  - Manipulation of data must be performed on operands already in processor registers.
  - Restriction may require additional instructions for tasks.
- However, it is possible to specify three operand instructions into a single 32-bit word, where all three operands are in registers:

Three-operand instruction


OP code
R*i*
R*j*
R*k*
Other info